LUNDS TEKNISKA HÖGSKOLA

Digitala Projekt, EITF40

MIDImega

Felix Hall, dt05fh0
handledare: Bertil Lindvall
2011-03-11

Denna artikel redogör för konstruktionen av ett digitalt musikintrument av typen sequencer. Hårdvaran är uppbyggd kring en ATmega16, och mjukvaran implementerades i C++. Artikeln beskriver planeringsfasen, för att därefter behandla de problem som påträffades. Slutligen ges en beskrivning av den slutgiltiga produkten.

Innehållsförteckning

Inledning

Kursen EITF40 vid LTH har som mål att studenterna under sju veckor ska färdigställa ett projekt, uppbyggt kring en mikrokontroller. Projektet ska inledas med en kravspecifikation och avslutas med en skriftlig rapport och en muntlig redovisning. Projektet ska liknas vid ett prototypförfarande i industrin, där ett proof-of-concept kan behöva tillverkas under kontrollerade former på kort tid.

Målet med mitt projekt var att tillverka en MIDI-sequencer, vilket är ett verktyg för att producera musik. Mer specifikt är en sequencer mjuk- eller hårdvara som lagrar rytmer och melodier, och kan återge dessa via exempelvis MIDI-protokollet. MIDI-protokollet är ett digitalt punkt-till-punkt-protokoll i vilket händelser, så som att en användare trycker på en knapp eller vrider på en ratt, kodas i paket och skickas seriellt till en mottagare. Denna mottagare använder sedan informationen i paketen för att skapa ljud. Projektet fick namnet MIDImega.

Arbetsmetod

Eftersom jag tidigare inte arbetat med särskilt mycket programmering på låg nivå och saknade erfarenhet och kunskap om standardlösningar på problem så började jag tidigt med att läsa igenom databladet för mikrokontrollern. Jag läste även in mig på andra, liknande projekt på avrfreaks.com. Under den första läsveckan producerade jag ett översiktligt kretsschema och en konceptbild över hårdvaran. Dessa följdes relativt väl, med några mindre tillägg.

Eftersom MIDImega innehåller väldigt mycket output, och även mycket input, så tog det lång tid att få all hårdvara på plats. I princip all hårdvara var inkopplad i läsvecka fyra, och den inledande testningen av visuell output var relativt tillfredsställande, då åtta av nio shiftregister fungerade som tänkt. Shiftregisterna används för att driva den stora mängden lysdioder utifrån några få pinnar på mikrokontrollern. Då jag fokuserat väldigt mycket på denna kurs, eftersom jag upplevde den som rolig, hade jag vid det här laget hamnat väldigt mycket efter i min andra, mindre intressanta kurs. Detta kompenserades för under läsvecka fem och sex, vilket ledde till att mjukvaruimplementationen av sequencern blev lidande. Mer om detta senare.

Kravspecifikation

Min kravspecifikation innehöll följande, mer eller mindre direkt formulerade krav:

MIDImega ska kunna styra följande visuella output:

- 24 LEDs för att ange vilken takt samt vilket taktslag som spelas upp eller redigeras
- 13 LEDs för att ange vilka toner som är aktiva
- 1 LED för att ange om MIDI-data spelas upp eller redigeras
- 3 siffror på en display för att ange parametervärde

MIDImega ska kunna ta emot och tolka följande input:

- 20 tryckknappar för val av funktion
- 1 vridbar pulsgivare

Dessutom ska MIDImega kunna sända MIDI-meddelanden utifrån förprogrammerade melodier, lagrade i minnet. (Denna punkt är något förändrad från den inledande kravspecen, då full funktionalitet för att redigera de programmerade melodierna inte har implementerats.)

Implementation

Hårdvara

ATmega16

Centralt i detta projekt är mikrokontrollern ATmega16. Av dess funktionalitet används följande:

PortA:

- Pins A0-A4 används när knappkodaren läses av. En binär representation av som mest 20 diskreta knappar är möjlig.

PortB:

- Pin B1 används för att skicka ut de bitar som ska skiftas in i skiftregistren.
- Pin B2 används för avbrottsförfrågan 2 från den roterande pulsgivaren.
- Pins B4-B7 är reserverade för att i framtiden kunna expandera mängden input med hjälp av SPI.

PortC:

Används uteslutande för JTAG-gränssnittet.

PortD:

- Pin D0 är reserverad för att i framtiden även kunna ta emot MIDI via USART.
- Pin D1 används för att skicka MIDI-meddelanden. Signalen buffras genom 74HC240 innan den via ett 220Ohms-motstånd leds till DIN-kontakten.
- Pin D2 används för avbrottsförfrågan 0 från knappsatsen.
- Pin D3 används för avbrottsförfrågan 1 från den roterande pulsgivaren.
- Pins D4-D7 används för att styra de nio skiftregisterna. En klocksignal för att skifta bitar genom registren och en för att flytta data från dataregistren till dess outputs (flip-flop). En signal för att tömma registren på data, och en för att tillåta att data matas ut.

Timer:

Timer0, en inbyggd 8-bitars timer, används för att avgöra när MIDI-data ska placeras i USARTs sändbuffert. Timern drivs i nuläget av en nerskalad systemfrekvens och genererar interrupts när ett visst värde, korrelerat till det tempo som anges av den interna variabeln `tempo`, uppnåtts.

Interrupt:

Samtliga externa interruptpinnar på mikrokontrollern används för avbrottsförfrågningar.

74HC240 inverterande buffer

Används för att driva MIDI från USART. Det är inte säkert att detta behövs, men det kändes tryggt att inte dra strömmen från mikrokontrollern. Eftersom signalen inte ska inverteras så inverteras den två gånger.

Den används även för att invertera signalen Data Available från knappsatskodaren, för att därefter styra dennas Output Enable.

Input

Knappsats med 12 tryckknappar

Då det hade blivit väldigt mycket att löda ifall alla 19 knappar skulle vara lösa så valde jag att använda en knappsats med tolv knappar. Dessa är organiserade i en brytarmatris, och raderna och kolonnerna drivs via virbara pinnar. Dessa kopplades till en knappkodare, en IC-krets som innehåller logik för att representera nedtryckta knappar binärt, och som även har inbyggda pull-up-motstånd.

5 lösa tryckknappar

Eftersom jag ville ha fler än de tolv knappar som fanns på knappsatsen, men ville ha dem annorlunda organiserade än hur det hade blivit om jag tagit en 20-knappsarray, så lade jag själv till fem lösa knappar. Om detta var ett klokt beslut, jämfört med att bara ta en färdig, större knappsats kan diskuteras. Jag integrerade även denna med knappkodaren. 12-knappsatsen var organiserad med tre kolonner gånger fyra rader. Logiskt på knappkodaren, som var organiserad enligt fyra kolonner gånger fem rader, så kopplade jag de fyra raderna som kolonner, och kolonnerna som rader. Då hade jag två rader fria. Till den ena av dessa ben för rader kopplade jag då det ena benet för fyra av de lösa tryckknapparna, och till det andra radbenet kopplade jag den 19e knappens ena ben. Därefter kunde jag från de andra benen på de lösa knapparna dra en ledare till var kolonn, för att därigenom kunna bygga en fullständig matris med samtliga knappar.

Knappkodare

Utöver de anslutningar som nämdes ovan så anslöts två kondensatorer till knappkodaren, mellan kbd-mask och jord, och mellan osc och jord. Dessa enligt kopplingsschema för asynkron dataöverföring, angett i knappkodarens datablad.

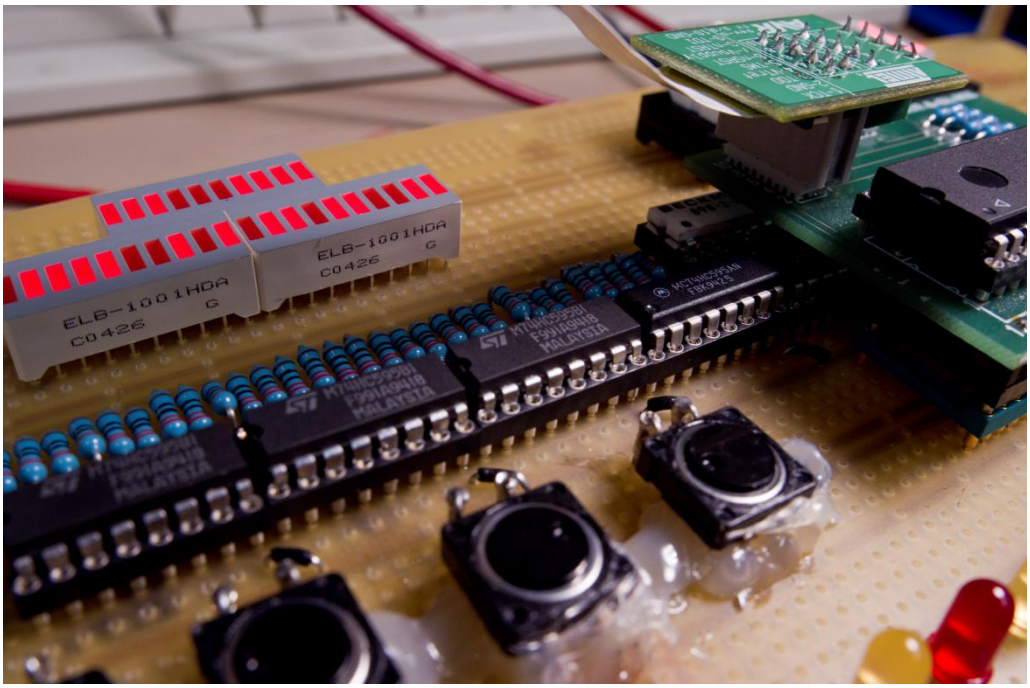
Dessutom anslöts pinnen Data Available till interrupt 0 på mikrokontrollern, och genom inverterare till output enable.

Med hjälp av ett oscilloskop tittade jag på hur de olika knapparna kodades och skrev kod i anslutning till interrupthanteringen för att avkoda dessa.

Roterbar pulsgivare

För att kunna justera parametrar valde jag att använda en pulsgivare. En sådan kan, till skillnad från en vridpotentiometer, snurras oändligt antal varv. Den har inte heller någon absolut position, utan digitala variabler kan ökas eller minskas beroende på antal snäpp en ratt snurras i en given riktning.

Pulsgivaren har tre ben, varav ett ansluts till VCC och de båda andra till varsin pinne på mikrokontrollern. För varje snäpp som ratten snurras så släpps ström antingen igenom, eller inte, på respektive ben. Ordningen i vilken pulser kommer på benen beror på vilket håll som ratten snurras på. Genom att ha avbrottsrutiner för båda benen, och i dessa läsa status på det andra benet, som inte utlöst avbrott, kan logik för att förändra variabler implementeras.



Output

74HC595 seriell-in/parallel-ut shiftregister med åtta bitar

Vid valet av metod för att styra 68 lysdioder fanns det lite att tänka på.

Att använda en pinne per lysdiod hade inte gått, även om jag använt alla pinnar på ATmega16 bara till dioderna, så det var uteslutet.

För att koppla lysdioderna i en matris hade det krävts 8+9 pinnar, vilket inte heller hade räckt till, med tanke på att jag ville reservera portar för framtida expansioner.

Jag började med att undersökta en metod som informellt kallas Charlieplexing. Den går ut på att utnyttja tri-state:ade utgångar, source, och sink, och genom ett sinnrikt mönster ansluta alla de lysdioder som ska drivas till pinnar på mikrokontrollern. Jag ansåg att det skulle bli väldigt svårt att felsöka ifall något skulle bli fel, och att det fortfarande skulle krävas för många pins från mikrokontrollern, så jag letade vidare.

Till slut insåg jag att den enklaste lösningen skulle bli att styra alla LEDar från skiftregister, i vilka jag klockar in bitar för vilka ben som ska aktiveras. För att använda minimalt antal pinnar på mikrokontrollern så kaskadkopplade jag de nio skiftregistren, så att den bit som klockas ut ur ett register i kedjan klockas in i nästa register. På så sätt klarade jag att styra alla 68 LEDs med fem pinnar, nämligen:

- En pin för att skicka ut själva datan
- En pin för att klocka in datan i registren
- En pin för att transportera den inklockade datan till registrens utgångar
- En pin för att tömma registren på data. Denna hade kunnat undvikas genom att inleda med att klocka in 72 nollor.
- En pin för att slå på registren. Denna hade också kunnat undvikas, då mjukvaran inte utnyttjar denna funktionalitet i nuläget.

7-segments LED-display
Egentligen 8 segment, då de också har en decimalpunkt. Använder tre stycken av dessa, och styr dem via skiftregistren. Strömbegränsande motstånd virade mellan skiftregistren och diplayens inputs.

Ska användas för att visa nuvarande parametervärde och annan information.

10-segments LED-display
Tre stycken, varav en används för att visa vilken takt som spelas eller redigeras, och två för att visa vilket taktslag. Även dessa med strömbegränsande motstånd.

Diverse lysdioder
För att markera vilka noter som är aktiva, samt ifall MIDI spelas upp eller inte.

MIDI-interface
Kopplat från USART Tx, via två inverterare (buffert) och ett motstånd till en pin i en DIN-5-hylskontakt.

Mjukvara

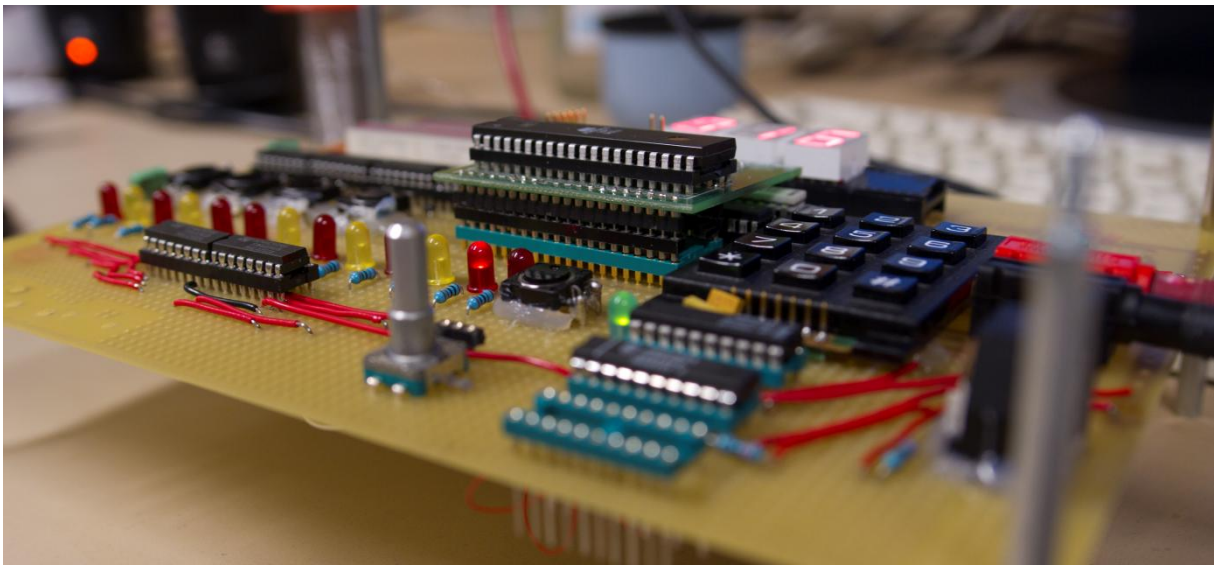
Input
Data utifrån läses vid interrupts. Tre interrupts är aktiva, varav ett för knappsatsen och två för den roterbara pulsgivaren.

Då en knapptryckning registreras av knappkodaren sätts Data Available hög, vilket triggar ett externt avbrott i mikrokontrollern. Avbrottsrutinen läser av A0-A4 och kodar av dessa med hjälp av förprogrammerade masker som representerar de olika knapparna. Lämpliga åtgärder kan sedan initieras, så som att byta nuvarande interna state.

När pulsgivaren snurrar genereras avbrott. Pulsgivaren ger pulser på två olika ben, och båda genererar avbrott. Med om detta i förbättringar nedan. Vid varje avbrott läses de båda pinnarna som är kopplade till pulsgivarens båda ben av, och det går därigenom att avgöra vilket håll pulsgivaren snurrats på.

Output
I programmets huvudloop görs avläsning av interna variabler och därifrån uppdatering av displayernas status. Skrivning till skiftregistren görs enligt proceduren beskriven under 74HC595 ovan.

Dessutom placeras MIDI-meddelanden i USARTs sändbuffert då ett timer-relaterat avbrott triggas. Detta görs med hjälp av GPL-registrerade avr-midi, vilket är ett bibliotek av vanliga MIDI-funktioner som wrappar USART-funktionaliteten i ATmega16.



Resultat

Den färdiga MIDImega uppfyller alla uttalade krav som ställts på den, även om den ännu inte går att använda som den MIDI-sequencer jag planerade från början.

Alla LED-displayer och att läsa data från knappsatsen fungerar utan problem. Pulsgivaren fungerar hjälpligt, på så sätt att den uppdaterar värden i rätt riktning beroende på vilket håll den snurrar på. Den hade dock behövt rutiner för att hantera kontaktstuds, vilket ännu inte implementerats.

Innan redovisningstillfället förväntar jag mig att två i förväg inprogrammerade melodier ska kunna spelas upp via MIDI. En hårdvarusynt kommer att medtagas till redovisningen.

Problem och svårigheter

Mitt första problem låg i att det var knappt fem år sedan jag läste min enda kurs i elektronik, vilket ledde till att jag kopplade avstörningskondensatorerna fel. Jag satte dem i serie med strömförsörjningen, istället för mellan V_{CC} och jord. Jag insåg dock snabbt mitt misstag, och flyttade dem.

Mitt nästa misstag låg i att jag tog för givet att virtråd vad för tunn för att leda strömmen som krävs för att tända lysdioder (2mA). Jag spenderade därför större delen av en helg med att löda små, små bitar grövre ledare. Detta var dock helt onödigt, fick jag senare veta.

Därefter gick arbetet relativt bra, och jag fick faktiskt output på åtta av de nio skiftregistren. Efter att ha felsökt koppligar på kretskortets kopplingssida, följt av ett par timmars felsökning i koden, tittade jag på signalen till det sista skiftregistret med hjälp av ett oscilloskop. Efter att ha insett att signalen såg ut att hoppa helt oberoende av vilken data som skickades in så insåg jag att isoleringen av en ledare på kortets ovansida hade smält och låg emot det sista skiftregistrets in-datapinne. Felet var snabbt åtgärdat medelst tång, och samtliga lysdioder tändes som planerat.

Trots viss tidigare bekantskap med språket C så uppstod som vanligt vissa svårigheter vid sammankomst med syntaxen vid användning av arrayer och pekare. Detta saktade ner utvecklingen av mjukvaran avsevärt.

Lärdomar
Som total nybörjare på denna nivå av både hårdvara och mjukvara har jag under denna kurs lärt mig fantastiskt mycket. Problemet jag tog mig an verkade, vid diskussioner med andra studenter som är mer erfarna inom området, vara mig övermäktigt, men jag är mycket nöjd med vad jag uppnått.

Diskussion

Vidareutveckling och förbättringar
Under denna rubrik hade jag kunnat skriva hur mycket som helst, med tanke på att jag vill kunna använda MIDImega till att experimentera med musikproduktion på riktigt. Ett par punkter kändes dock viktigare än andra.

Reset-knapp
Då mjukvaran för tillfället inte är stabil hade det varit bra med en reset-knapp, för att snabbt återställa en krashad MIDImega till dess ursprungstillstånd.

Fungerande programmeringsinput
I dagsläget kan MIDImega endast spela upp i förväg inprogrammerade melodier. I framtiden planerar jag att utveckla ett väl fungerande gränssnitt för att programmera melodier och spara dessa i EEPROM.

Avstudsning
Pulsgivaren genererar mycket kontaktstuds, vilket gör att det avlästa värdet kan hoppa flera steg i någon riktning. Att implementera en avstudsningsrutin borde inte vara svårt.

Kapplöpningsproblem
I dagsläget finns det fyra avbrottsrutiner och en huvudloop som behandlar gemensam data. Datan är deklarerad `volatile`, men jag har inte undersökt ifall en del funktionalitet borde göras atomiskt.

Använda polling istället för interrupt
På grund av att de mekaniska brytarna på konstruktionen inte går att få studsfria hade det varit bra att använda polling istället för avbrott vid avläsning av input. Detta hade antagligen gett mjukare respons på input än den nuvarande implementationen. Det hade också förhindrat många av de kapplöpningsproblem som för tillfället kan inträffa.

Framtiden
Jag planerar att fullfölja mina ursprungliga intentioner och tillverka en fullt fungerande sequencer. Utvecklingen av mjukvara tog längre tid än planerat, men passar bra att göra hemma på fritiden. Jag har införskaffat en AVR Dragon med allt som krävs för JTAG-anslutning, och har lagt till en spänningsregulator för att kunna använda en vanlig batterieliminator.

Appendix – källkod

Koden är under utveckling, och jag har planer på att refaktorisera den. För tillfället ligger all kod i samma fil, och koden är i ett stadie mitt emellan interrupt- och pollingstyrd input. Den har dessutom nyligen övergått till en annan lösning för timer. Vid intresse, titta på källkoden via länkar på hemsidan istället.

`midimega.c`

```
#include <avr/io.h>
#include <inttypes.h>
#include <stdbool.h>
#include <avr/interrupt.h>
#include "midi.h"
#include "midi.c"

void clearAllDisp();
void clearBars();
void clearBeats();
void clearNotes();
void clearSRs();

void refreshDisplay();

void edit();
void stop();
void play();
void play_note(int);

void tick();
void sbeat();            // sixteenth_beat
void qbeat();            // quarter_beat
void hbeat();            // half_beat
void nextbeat();         // one beat, 1/16 note

void readInputs();

void setDispStatus();
void barLed(int, int);
void noteLed(int, int);
void greenLed(int);
void setDispNum(int);
void digit(int, char);
void led(int, int);

void srclk(int);
void rclk(int);

void read_kbd();
void encoder_cw();
void encoder_ccw();

#define MIDI_CLOCK_RATE 15

#define C1   13
#define C2   25
#define C3   37
#define c    0
#define db   1
#define d    2
#define eb   3
```

```
#define e    4
#define f    5
#define gb   6
#define g    7
#define hb   8
#define h    9
#define ab   10
#define      a            11
#define b    12
#define pause             -1

volatile int state;                  // 0 = stop, 1 = play, 2 = edit

volatile int tempo;
volatile int ticks;
volatile int current_beat;
volatile int nbrbeats;
volatile int nbrbars;

volatile int current_octave;

volatile int bh_current;
volatile int buttonHistory[] = {0,0,0,0,0};

volatile int rh_current;
volatile int rotaryHistory[] = {0,0,0,0,0};

volatile unsigned char dispdata[9];

unsigned int notes[] = {  c, c, g, g, a, a, g, pause,
f, f, e, e, d, d, c, pause,
g, g, f, f, e, e, d, pause,
g, g, f, f, e, e, d, pause,
c, c, g, g, a, a, g, pause,
f, f, e, e, d, d, c, pause };

/*

PortA:
pa0: kbd
pa1: kbd
pa2: kbd
pa3: kbd
pa4: kbd
pa5: -
pa6: -
pa7: -

PortB:
pb0: usart clock
pb1: Display data out
pb2: Encoder interrupt, INT2
pb3: -
pb4: spi
pb5: spi
pb6: spi
pb7: spi

PortC: JTAG

PortD:
pd0: MIDI In
```

```
pd1: MIDI Out
pd2: kbd interrupt, INT0
pd3: Encoder interrupt, INT1
pd4: Shift clock
pd5: Storage clock
pd6: Shift registers output enable
pd7: Shift registers clear

*/

int main(void)
{

            DDRA        =           0x00;       // All pins for input
            DDRB        =           0x02;       // Set PINB1 for disp data output,
rest input
            DDRD        =           0xF0;       // Set PIND4-7 for disp control,
rest input
            PORTA       =           0x00;       // Disable pullups
            PORTB       =           0x04;       // Disable pullups, except INT2
            PORTD       =           0x08;       // Disable pullups, except INT1

            // Init timer, compare A
            TIMSK |= 0x10;
            TCCR1B |= 0x04;

            clearSRs();                         // Clear shift registers
            clearAllDisp();                     // Init dispdata

            PORTD       |=          0x80;       // Enable output

            /*
            MCUCR       |= 0x0F;    // Set trigger conditions for INT0 and INT1
            MCUCSR      |= 0x40;    // Enable trigger for INT2
            GICR        |= 0xE0;    // Enable external interrupts 0, 1 and 2
            */

            sei();                                          // globally enable
interrupts

            midiInit(MIDI_CLOCK_RATE, true, false);  // Initialize midi


            // State info
            state = 0;
            ticks = 0;

            // Init input histories
            bh_current = 0;
            rh_current = 0;

            current_octave = 3;

            // Beat config and init
            tempo                   = 120;
            nbrbeats    = 16;
            nbrbars     = 8;
            current_beat = 0;

            // Set timer compare to 7324/tempo
            // 7324 is the amount of clock ticks for one beat per minute
            OCR1A = 0x00+(7324/tempo);
```

```
		while(1) {
			readInputs();

			if(state == 1){
				TCCR1B |= 0x04;
			}
			else TCCR1B &= ~0x04;



			setDispStatus();
			refreshDisplay();
		}

		return 0;
}

void readInputs()
{


}

/* Start of beat handling */

void tick()
{
		ticks++;
		if(ticks%16 == 0) {
			nextbeat();
			ticks = 0;
		} else if (ticks%8) {
			hbeat();
		} else if (ticks%4) {
			qbeat();
		} else if (ticks%2) {
			sbeat();
		}
}

void sbeat()
{
}

void qbeat()
{
}

void hbeat()
{
}

void nextbeat()
{
		current_beat++;
		if(current_beat >= (nbrbars*nbrbeats))
			current_beat  = 0;
}

void prevbeat()
{
```

```
            current_beat--;
            if(current_beat < 0)
                        current_beat = (nbrbars*nbrbeats);
}

/* End of beat handling */



/* Start of LED-lighting routines */

/*
0-19 = beats
20-29 = bars
30-37 = digit1
39-45 = digit2
47-53 = digit3
54-55 = UNUSED
56-68 = notes
69-70 = UNUSED
71    = power
*/

void setDispStatus()
{
            if(state == 1)
                        greenLed(1);
            else
                        greenLed(0);

            clearBars();
            barLed( (current_beat/nbrbeats)%nbrbars , 1);

            clearBeats();
            led(current_beat%nbrbeats, 1);

            if(state == 0) {
                        digit(1, 'o');
                        digit(2, 'f');
                        digit(3, 'f');
            } else if (state == 1) {
                        digit(1, 'p');
                        digit(2, 'l');
                        digit(3, 'A');
            } else if (state == 2) {
                        digit(1, 'e');
                        digit(2, 'd');
                        digit(3, 'i');
            }
}

// Higher level routine for toggling bar leds. 0 is first.
void barLed(int nbr, int on)
{
            led(nbr+20, on);
}

// Higher level routine for toggling note leds. 0 is first.
void noteLed(int nbr, int on)
{
            led(nbr+56, on);
}
```

```
// Higher level routine for toggling the green led.
void greenLed(int on)
{
		led(72, on);
}

void led(int nbr, int on)
{
		int graph = (int)((nbr)/8);

		if(on){
			dispdata[graph] |= 1 << (7-(nbr%8));
		} else{
			dispdata[graph] &= ~(1 << (7-(nbr%8)));
		}
}

/* End of LED-lighting routines */

/* Start of routines for handling 7seg displays */

// Higher-level routine for setting the three 7seg displays to a positive int value
<1000
void setDispNum(int i)
{
		int d1 = i/100;
		int d2 = (i-d1*100)/10;
		int d3 = i-d1*100-d2*10;
		digit(1, (char)('0'+d1));
		digit(2, (char)('0'+d2));
		digit(3, (char)('0'+d3));
}

void digit(int digit, char letter)
{
		unsigned char mask;
		switch (letter)
		{
			//TODO: Skriv om till enum, snabbare och mindre kod
			case '0':
				mask = 0b10111110; break;
			case '1':
				mask = 0b00100010; break;
			case '2':
				mask = 0b01011110; break;
			case '3':
				mask = 0b01110110; break;
			case '4':
				mask = 0b11100010; break;
			case '5':
				mask = 0b11110100; break;
			case '6':
				mask = 0b11111100; break;
			case '7':
				mask = 0b00100110; break;
			case '8':
				mask = 0b11111110; break;
			case '9':
				mask = 0b11110110; break;
			case 'a':
			case 'A':
```

```
                                        mask = 0b11101110; break;
                        case 'b':
                        case 'B':
                                        mask = 0b11111000; break;
                        case 'C':
                                        mask = 0b10011100; break;
                        case 'c':
                                        mask = 0b01011000; break;
                        case 'd':
                        case 'D':
                                        mask = 0b01111010; break;
                        case 'e':
                        case 'E':
                                        mask = 0b11011100; break;
                        case 'f':
                        case 'F':
                                        mask = 0b11001100; break;
                        case 'H':
                                        mask = 0b11101010; break;
                        case 'h':
                                        mask = 0b11101000; break;
                        case 'i':
                        case 'I':
                                        mask = 0b00100011; break;
                        case 'j':
                        case 'J':
                                        mask = 0b00111010; break;
                        case 'l':
                        case 'L':
                                        mask = 0b10011000; break;
                        case 'o':
                        case 'O':
                                        mask = 0b10111111; break;
                        case 'p':
                        case 'P':
                                        mask = 0b11001110; break;
                        case 's':
                        case 'S':
                                        mask = 0b11110101; break;
                        case 'U':
                                        mask = 0b10111010; break;
                        case 'u':
                                        mask = 0b00111000; break;
                        default:
                                        mask = 0x00;
                }

                dispdata[digit+2] &= 0b11111100;
                dispdata[digit+2] |= mask >> 6;
                dispdata[digit+3] &= 0b00000011;
                dispdata[digit+3] |= mask << 2;

}

/* End of routines for handling 7seg displays */


// Outputs all dispdata into shift registers
void refreshDisplay()
{
                rclk(0);
                for(int i = 8; i>=0; i--) {
```

```
                        for(int j = 0; j<8; j++) {
                                srclk(0);
                                if( (dispdata[i] & (1 << j)) == (1 << j))
                                        PORTB |= 0x02; //Put 1 on PB1
                                else
                                        PORTB &= ~0x02;          //Put 0 on
PB1
                                srclk(1);
                        }
                }

                rclk(1);                                        // Toggle storage of
shifts
}

/* Start of routines for clearing diplays */

void clearAllDisp()
{
                dispdata[0] = 0x00;
                dispdata[1] = 0x00;
                dispdata[2] = 0x00;
                dispdata[3] = 0x00;
                dispdata[4] = 0x00;
                dispdata[5] = 0x00;
                dispdata[6] = 0x00;
                dispdata[7] = 0x00;
                dispdata[8] = 0x00;
}

void clearBars()
{
                dispdata[2] &= 0xF0;
                dispdata[3] &= 0x03;
}

void clearBeats()
{
                dispdata[0] = 0x00;
                dispdata[1] = 0x00;
                dispdata[2] &= 0x0F;
}

void clearNotes()
{
                dispdata[7] = 0x00;
                dispdata[8] &= 0x01;
}

// Inits all shift registers
void clearSRs()
{
                PORTD       =           0x08;       // Disable pullups, except INT1
                srclk(0);
                srclk(1);
                rclk(0);
                rclk(1);
}

/* End of routines for clearing displays */
```

```
/* Clocking routines */

void srclk(int i)
{
		if(i == 1)
				PORTD |= 0x10;
		else
				PORTD &= ~0x10;
}

void rclk(int i)
{
		if(i == 1)
				PORTD |= 0x20;
		else
				PORTD &= ~0x20;
}

/* End of clocking routines */


/* Start of input decoding routines */

void encoder_cw()
{
		if(state == 2) nextbeat();
		else current_octave++;
}

void encoder_ccw()
{
		if(state == 2) prevbeat();
		else current_octave--;
}

void read_kbd()
{
		unsigned int button;
		unsigned char kbdData = (PINA & 0x1F);
		switch(kbdData)
		{
				case 0b00000111:
						button = 0; break;
				case 0x00:
						button = 1; break;
				case 0b00000100:
						button = 2; break;
				case 0b00001000:
						button = 3; break;
				case 0b00000001:
						button = 4;  break;
				case 0b00000101:
						button = 5;  break;
				case 0b00001001:
						button = 6;  break;
				case 0b00000010:
						button = 7;  break;
				case 0b00000110:
						button = 8;  break;
				case 0b00001010:
						button = 9;  break;
				case 0b00000011:
```

```
                                button = 10;break;      // *
                        case 0b00001011:
                                button = 11;break;      // #
                        case 0b00001100:
                                button = 12;break;      // b1
                        case 0b00001101:
                                button = 13;break;      // b2
                        case 0b00001110:
                                button = 14;break;      // b3
                        case 0b00001111:
                                button = 15;break;      // b4
                        case 0b00010000:
                                button = 16;break;      // b5
                        default:
                                button = 99;break;
                }

                switch(button)
                {
                        case 0: play_note(10); break;
                        case 1:
                        case 2:
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 7:
                        case 8:
                        case 9:
                        case 10:
                                play_note(button-1); break;
                        case 11: play_note(11); break;
                        case 12: play(); break;
                        case 13: stop(); break;
                        case 14: break;
                        case 15: break;
                        case 16: edit(); break;
                        case 99:
                                break;
                }

}

/* End of input decoding routines */

void edit()
{
                state = 2;
}

void play()
{
                if(state == 1) {
                        // already playing
                } else {
                        state = 1;
                        current_beat = 0;
                }
}

void stop()
{
```

```
            state = 0;
}

void play_note(int i)
{
            midiSendNoteOn(0, current_octave*12+i, 127);
}

/* Start of interrupt routines */

// Keyboard data available
ISR(INT0_vect)
{
            read_kbd();
}

// Rotary encoder, pinA
ISR(INT1_vect)
{
            if(PIND & 0x08){
                        if(!(PINB & 0x04)){
                                    encoder_cw();
                        } else{
                                    encoder_ccw();
                        }
            } else {
                        if(PINB & 0x04){
                                    encoder_cw();
                        } else {
                                    encoder_ccw();
                        }
            }

}

// Rotary encoder, pinB
ISR(INT2_vect)
{
            if(PINB & 0x04){
                        if(PIND & 0x08){
                                    encoder_cw();
                        } else {
                                    encoder_ccw();
                        }
            } else {
                        if(!(PIND & 0x08)){
                                    encoder_cw();
                        } else {
                                    encoder_ccw();
                        }
            }
}

ISR(TIMER1_COMPA_vect)
{
            tick();
            TCNT1 = 0x0000;
}

/* End of interrupt routines */
```